CREATION OF A LINEAR OPTIMIZATION FILE FORMAT FOR INSTRUCTIONAL PURPOSES

Submitted to
The Engineering Honors Committee
119 Hitchcock Hall
College of Engineering
The Ohio State University
Columbus, Ohio 43210

By

Richard Anthony Cavolo, Jr.
325 W. 8th Ave
Columbus, OH 43201

May 20, 2005

Acknowledgements

I would like to thank my thesis advisor, Dr. R. Allen Miller, for his guidance and contributions to this project, Dr. Marc Posner and Dr. Clark Mount-Campbell for their insights and for teaching me mathematical programming, and all my teachers in the Industrial, Welding, and Systems Engineering department for their contributions to my education. In addition, I would like to thank my parents for their emotional and fiscal support.

Table of Contents

1.0 Abstract

Every year, over one hundred Industrial and Systems Engineering at Ohio State students take courses that teach them the basics of linear and integer programming. Currently, students in these courses are taught to use a built in Excel solver or CPLEX using the LP or MPS file formats. These formats make it frustrating for students to create, modify, and debug mathematical programs. This frustration can prevent students from learning. This paper suggests the use of a high-level Excel specification format that is easier to use and more powerful than the formats that are currently taught. An Excel add-in was created to convert this new file format into an industry standard format.

The new format and conversion program have many features to help students create, modify, read, and debug their models. Each section of the model has a column for a description and many different ways to add comments and notes exist. In addition, all of Excel's features can be used to derive the end model. Finally, an error checker that gives specific feedback and suggestions for fixing errors is built into the file converter. The new format still needs to be extensively tested and some features are yet to be added. In addition, testing should be done to determine how the use of the new format influences the learning of mathematical programming.

2.0 Introduction

Every year, over one hundred Industrial and Systems Engineering at Ohio State students take courses that teach them the basics of linear and integer programming. These basic industrial engineering skills form the basis for theory in other courses such as facility layout and scheduling. Many students struggle with the distinction between variables and parameters. In addition, debugging tasks are extremely difficult since every coefficient needs to be individually calculated and added to the model. If a file format for specifying mathematical programs is well designed, its structure may help students learn mathematical programming techniques better than if they used more difficult file formats. In addition, a file format that handles the repetitive aspects of model creation may decrease the time spent debugging and refining models.

Mathematical programming is a set of methods that are used to make decisions to optimize some result. In mathematical programming, the real world is modeled as a set of linear equations. One equation is designated as the objective function, which is either maximized or minimized by changing the variables in that function. Other functions, called constraints, are used to define a convex solution space. These functions place limits on the variables in the function and allow the user of mathematical programming to model the real world. If the variables used are allowed to assume continuous values, the problem is referred to as a linear programming problem. Likewise, problems with all variables constrained to integer values are called integer programming problems. Finally, problems with both integer and continuous variables are called mixed integer programming problems. Mathematical programs are usually solved by computer and

many different computer input formats are available. An example linear program to maximize profit in a cheese factory is included below.

Cheese Factory Production Model

Subscripts

$i \in \{1,2,3\}$ = the type of cheese

Parameters

D_i = Demand for Cheese Type i

P_i = Profit made by making and selling 1 ton of Cheese Type i

Variables

C_1 = Tons of Cheese Type 1 Produced

C_2 = Tons of Cheese Type 2 Produced

C_3 = Tons of Cheese Type 3 Produced

Objective Function

$Max: Z = \sum_i P_i * C_i$ Total profit

Constraints

$C_i \leq D_i \quad \forall i$ Limits the cheese produced to demand

$2 * C_1 + C_2 \leq 65$ Limit based on amount of cheese cloth available

$C_1 + C_2 + C_3 \leq 80$ Limits based on amount of milk available

In the above model, the three decision variables, C_1, C_2, and C_3 will be set to values that maximize the objective function without violating the constraints. The parameters D and P are set by the user to real numbers based on current demand and current profit of cheese. This sort of model might be run every month to set production targets for the month. In addition, the use of subscripts and parameters can make the model scalable. If the last two constraints were written so that they used parameters and

the variables were written using the subscript *i*, the entire model could be changed to accommodate another type of cheese by merely changing the range of the subscript *i* and adding new parameter values.

At The Ohio State University, undergraduate students are currently taught to use Excel and CPLEX. Excel has a built in solver that is unreliable for integer models and is constrained by the number of columns in Excel. This format is only practical for small, simple models. CPLEX is taught with two different file formats, LP and MPS. In the LP file format, users write each constraint on a separate line with coefficients placed before each variable name. The right hand side is specified on the same line as the constraint. Additional constraints on variable values are placed at the end of the file. In the MPS file format, separate sections give the row sense of each constraint, the coefficients of each variable in each constraint, the row sense, and bounds on the values that variables can take. This format is easier to create via code. Both file formats are industry standards and can be solved with a host of solver suites.

3.0 Project Objectives

The goal of this thesis is to create a file conversion program that converts a new, well structured linear programming file format into an industry standard format. The purpose of this new format is to provide students with a freely distributed file converter that teaches through its structure and speeds up the development of large, repetitive models. This file format is based on research regarding the ways that students learn mathematical concepts and programming skills. In addition, the format is designed to help students debug and modify models.

4.0 Project Approach

A literature review was conducted to understand the process by which students learn and utilize mathematic and programming knowledge. In addition, an existing modeling language, AMPL, and existing mathematic programming file formats, MPS and LP, were reviewed for conflicts with current theory on mathematical and programming learning processes.

The information gained from this review was used to define the characteristics of an input format ideal for the teaching and learning mathematical programming. These characteristics were weighted against basic file types to determine that an Excel based solution was the best available option.

After Excel was chosen, a breakdown of a generic high level mathematical programming format was laid out. An Excel add-in was then developed to translate this high-level format into the MPS format for exporting into an existing solver.

The model was tested using a basic model derived from an integer programming exam that incorporated all of the features of the initial program version. In addition, the cheese production model, used as an example in this paper, was also tested. A test user will be used to identify programming errors and usability deficiencies in the near future.

Finally, this paper, which describes the intended use and reasoning behind the design of the file format, was written. This paper includes appendices with user documentation and a description of the error checking provided with the first version of the program.

5.0 Background Knowledge

Solving linear programs is difficult because the successful design and implementation of a solution involves the creative use of mathematical principles to create a mathematic model of reality and the subsequent entering of that model into a computer language so that a computer can solve for the optimum decisions to be made. In this way, linear programming requires a mix of skills borrowed from both mathematics and computer science. This paper will attempt to evaluate the process of learning to create linear programs from both the mathematical and computer science perspectives to better understand what characteristics are needed in an input format to help students through the phases of model design and implementation.

5.1 Notation and Syntax

The use of mathematic notation is closely tied to the understanding of students of mathematics and a lack of understanding can hinder the learning of higher mathematics. The ability to understand both the mathematic notation used and the underlying meaning of that notation is one key difference between successful students of mathematics and unsuccessful students. Furthermore, this understanding is closely tied to an internal sense of authority that most successful students seem to possess (Alcock et al, 2005). When first learning a new mathematical sign, students use that sign in problem solving before understanding that sign's cultural meaning (Berger, 2004). In fact, it has been suggested that most people are able to perform logic tasks when those tasks are presented in plain English rather than in formal mathematic notation (Morris, 2002). Clearly, notation in mathematics can hamper the successful acquisition of mathematic knowledge.

Similarly, the syntax of programming languages can prove to detrimental to novice programmers. An overly simplified structure can force undue restructuring of programs to achieve relatively simple results. In contrast, having too many options can also hinder learning since students still have to deal with the semantics of programming languages that are designed with advanced constructs in mind (McIver et al, 1996). Poorly designed syntaxes can hinder a student's ability to learn the underlying concepts that are vital to the development of good programming skills.

5.2 Importance of Readable Examples

Worked examples, both in lecture and in print, are an important part of learning mathematical concepts. The intricate and detailed knowledge that may be gained by the viewing of worked examples is a critical difference between experts and novices in a wide range of fields (Sweller, 1989). Sweller argues that viewing well formatted, worked algebra problems may increase algebraic problem solving ability more than the working of algebraic problems. This, he explains, results from the focus of a student's attention; specifically, a student that is working a problem is focused on finding a solution rather than understanding the underlying mathematical principles while a student reading a worked problem directs his or her undivided attention to the understanding of the reasoning behind the method used to solve the problem.

Similarly, anecdotal evidence exists that both novice and experienced students who learn to read programs before learning to write them are less likely to drop a programming class (Patton). While these claims need to be scientifically substantiated,

they provide further evidence that the readability and availability of sample work is a necessary component of successful learning of mathematic and programming structures.

5.3 Importance of Consistency

McIver and Conway describe in detail many of the inconsistent structures that exist in many common languages. In general, when constructs are confusing, mean different things in different situations, and/or do not exhibit the behavior that students expect, the learning of programming concepts is hindered by the language hardships that students face (McIver et al, 1996). The frustration that students can feel is further compounded by poor error checking, another important feature of languages that McIver and Conway discuss. Many languages have terse, unintelligible error messages that do not guide the user towards a meaningful resolution of programming errors. For example, the course packet for Computer Science 360 at The Ohio State University contains pages of advice for fixing various programming errors that might be encountered in ISEM, an instructional assembly level language.

6.0 Adaptation to Teaching Linear Programming

6.1 Syntactical Implications

Linear programming, as it is taught at Ohio State, is a very structured set of mathematical concepts. Thus, it is possible to use a relatively simple, high level input structure and still allow for enough flexibility that models can be entered without undue hardship. In order to avoid the problems associated with complex syntaxes and/or notation, the language or input structure should be as close to English as possible while

maintaining a close resemblance to the concise mathematic notation that is used to write out linear programming models for reading. LP and MPS file formats use very accessible formats that allow for easy entry. AMPL uses a programming style structure to form models that may have a familiar feel for programmers (Fourer et al, 2003). While its structure is relatively good blend of English and mathematical notation, the language has too many features and may be overwhelming for a novice student.

6.2 Readability

While MPS and LP models are syntactically simple, they are not as easy to read as AMPL. MPS places at most 2 coefficients on a single line and requires knowledge of the problem to be successfully read. Similarly, the LP format, while easier to understand, shows the variable names and their corresponding coefficients with no comments and thus requires background knowledge to be read. AMPL provides a much more readable format but requires more extensive knowledge of programming to be read.

6.3 Error Checking

Error checking plays a vital role in the successful and painless input of a mathematical model into a computer. A good compiler will give detailed and meaningful feedback on the input syntax and model validity to aid the successful development of a computer model. The LP and MPS format have no encapsulated error checking but rather rely on the solver to provide error messages to the user. AMPL provides good feedback with its error checker but does not always provide suggested fixes to errors.

7.0 Description of Final Work Product

7.1 Model Format Considerations

Two formats for the creation of models were considered before program development was started. The first design concept called for a text file with clearly marked sections for subscripts, parameters, variables, the objective, and constraints. A compiler would then be created to convert the text file to an industry standard model file format, such as MPS. This style solution would require a fairly rigorous syntax to make the compiler creation possible in the time allotted for this project. In addition, the user would be forced to remember the structure of the file and would start each model from a blank document. Since text files are standard, no additional programs would be needed for a student to use this sort of solution. In addition, such a file format could closely resemble industry used modeling languages.

In contrast, the second design concept called for an Excel workbook with separate worksheets for each element of the model. An Excel Add-In would then be created to aid model development and convert the information on the various sheets to an industry standard file format. Excel is a program that, based on anecdotal evidence, many students are both familiar and comfortable with. In addition, models built in Excel have the benefit of the built in structure of columns and rows. Finally, having an Excel based model allows users to create model values through use of formulas and other Excel features. Although access to Excel, a proprietary program, is more restricted than access to text editors, students generally have access to Excel in computer labs and can purchase the program at a reduced rate. After comparing the relative benefits and drawbacks of

each option, Excel was chosen as the overall environment in which to create a mathematical program development aid.

7.2 Components of the Model Workbook

In order to create a clear cognitive distinction between subscripts, parameters, variables, and equations, these elements appear in separate worksheets in the model workbook. The format for these worksheets is created via a menu option so that the user does not need to remember the format for the entire model. Subscripts and parameters have separate sheets for the declaration and definition of values. This allows users to scale existing models quickly and with decreased risk of errors. In addition, the definition pages can be made for a generic problem and distributed widely. Students can then add the subscript interpretation and parameter values into the pre-existing model and use it to solve problems. Finally, the add-in creates a separate sheet with the MPS model in it. This allows the user to review the model and make changes before creating the text file that will be opened with a mathematical programming package.

7.3 Key Usability Features

As mentioned earlier, the program allows for the creation of the model via formulas. Users can create powerful models that utilize Excel's features to pull in data from outside sources and calculate parameter values. In addition, by not entering data in column A on a particular line, users can add comments into the model that are not read by the model reading program. Also, users can use columns not explicitly used in the model to perform intermediate calculations and add notes. These columns are never touched by

the programs in the add-in. Finally, an error checker that provides specific feedback is provided where possible. This allows students to focus on the creation of their model rather than the structure of the file being created. An example error message is:

> An error occurred while processing the equation on row 11: An invalid syntax was found in the equation definition. Two variables, "TD" and "X[1, 1]", are being multiplied together. In order to be a valid linear model, variables cannot be multiplied together. Note: the substitution of subscripts has occurred and variables with subscripts will have been changed.

This error message gives the user the specific row that the error occurs, describes the error in detail, and lets the user know why the error is an error. See Appendices 2 and 3 for user documentation and an example model.

8.0 User Testing

User testing has not been completed as of the writing of this paper. Since user testing and study of use is an important part of the design process, this step will be completed soon.

9.0 Future Steps

The program is currently functional but still needs to be rigorously tested for programming errors. In addition, in depth user studies need to be done to improve the cognitive elements of the design. While the final product is based on research in mathematics education and expert opinion in computer science, research should be done to determine what effects, if any, the use of a model formation aid geared to novice users will have. This study should ideally use a class taught with current software and another class taught with this new software available. Each class should then be compared for a

general understanding of mathematical programming (as measured by test and homework performance), attitude towards mathematical programming, use of learned methods in later problem solving, and retention of skills a year or more after mathematical programming coursework is completed.

Some additional functionality should be added to enhance the student experience and allow for the easy creation of as many model types used in Industrial and Systems Engineering coursework as is possible. First, support for the indexing of subscripts via the addition or subtraction of a constant should be added to expedite the creation of scheduling and other periodic models. Next, the addition of warnings to notify users that a subscript, parameter, and/or variable was declared and not used will help users detect typing errors and easily see if they utilized all of the model elements that were declared. Also, a program that reformats the model in an easy to read, printable format will help students debug their models and, if done correctly, can be the format that is used to submit models to professors. Moreover, a list of all equation and variable names with a list of the particular subscript values that that variable or equation uses will help students read the output file from the optimization program that they use. In addition, an output format for exporting the model into Storm may be useful. Storm has an optimization package that uses a tableau format. The tableau format may also have other uses for students and educators. Finally, a help file should be created that makes information on the add-in readily available to users.

10.0 Conclusion

The ability to formulate mathematical programs is valuable skill that many students have difficulty in developing. By using a modeling aid specifically designed to help students create mathematically programs, students may benefits from increased understanding of mathematical programming concepts, a higher opinion of mathematical programming, and increased retention of understanding and knowledge. Well written error messages allow the user to focus on the creation of a model and worry less about the formatting and finding of errors. In addition, the readable and scalable format can be used by professors to give models to students in a format that they can both study and use in other classes.

The Excel add-in that was developed comes close to accomplishing many of these goals. Further testing and some additional development are needed to allow the program to reach its full potential. In addition, usability studies need to be done to measure how effective the new file format is in the facilitation of the learning of mathematical programming.

References

Alcock, L., & Simpson, A. (2005). Convergence of sequences and series 2: interactions between nonvisual reasoning and the learner's beliefs about their own role. *Educational Studies in Mathematics, 58*(1), 77-100.

Berger, M. (2004). The functional use of a mathematical sign. *Educational Studies in Mathematics, 55*(1-3), 81-102.

Fourer, R., Gay, D., & Kernighan, B. (2003). *AMPL: a modeling language for mathematical programming.* 2nd ed. Pacific Grove, CA: Brooks/Cole - Thomson Learning.

McIver, L. & Conway, D. M. (1996). Seven sins of introductory programming language design. Retrieved April, 2005, from Damian Conway's Online Papers Web site: http://www.csse.monash.edu.au/~damian/papers/.

Morris, A. K. (2002). Mathematical reasoning: adults' ability to make the inductive-deductive distinction. *Cognition and Instruction, 20*(1), 79–118.

Patton, P. C. (n.d.). New methods for teaching programming languages to both engineering and computer science students. Retrieved April 2005, from http://www.micsymposium.org/mics_2004/Patton.pdf.

Sweller, J. (1989). Cognitive technology: some procedures for facilitating learning and problem solving in mathematics and science. *Journal of Educational Psychology, 81*(4), 457-466.

Appendix 1: Project Schedule

The following table contains an approximate timeline on which the project was completed. All activities were shortened in duration to reflect a correction in the Oral Defense deadline. As a result of this shortening, the number of test files created was shortened from 3 to 1. In addition, errors were introduced in the test file instead of creating separate error files. Error checking was built into the compiler as a result of the object oriented style used and an overlapping of programming constructs required for each task.

Project Schedule

#	Task	Start Date	Finish Date	Pred
1	Review LP Languages	3/28/2005	4/8/2005	
2	Review Teaching Language Literature	3/28/2005	4/8/2005	
3	Consult OR Faculty	3/28/2005	4/8/2005	
4	Develop File Format Specifications	4/11/2005	4/15/2005	1,2,3
5	Develop a Test File w/out Errors	4/18/2005	4/22/2005	4
6	Create the Compiler	4/18/2005	5/6/2005	4
7	Test Compiler Using Test Files	5/9/2005	5/13/2005	6,5
8	Develop a List of Errors to Check for	4/11/2005	4/15/2005	1,2,3
9	Create the Error Checker	4/18/2005	5/6/2005	8
10	Test the Error Checker Using Modifications to the Test File	5/6/2005	5/9/2005	9
11	Integrate Compiler and Error Checker	5/9/2005	5/11/2005	7,10
12	Test the Integrated Program	5/11/2005	5/12/2005	11
	Oral Defense Deadline	5/13/2005		
14	Create User Documentation	4/18/2005	5/20/2005	
15	Create Document Explaining the Rationale for the Project	4/18/2005	5/20/2005	
	Written Thesis Deadline	5/20/2005		

Appendix 2: User Documentation

A2.1 Installation

To install the Excel Add-in referred to in this paper, first open the file. Next, navigate to the "Instructions" worksheet. Follow the instructions that appear in that worksheet to install the add-in. The add-in will remain active until it is deselected in the add-ins pane located in the Tools menu.

A2.2 General Knowledge

When creating models, a few features are the same on all model definition worksheets (the six worksheets on which model components are defined). The compiler only reads rows that have a value in column A on all worksheets. This means that comments and spacing can be added to worksheets by simply leaving the first cell of any line blank.

In addition, any column that does not have a column heading in the template generated by the add-in is ignored by the compiler. This means that all of the columns to the right of the model can be used for comments, intermediate calculations, and data storage.

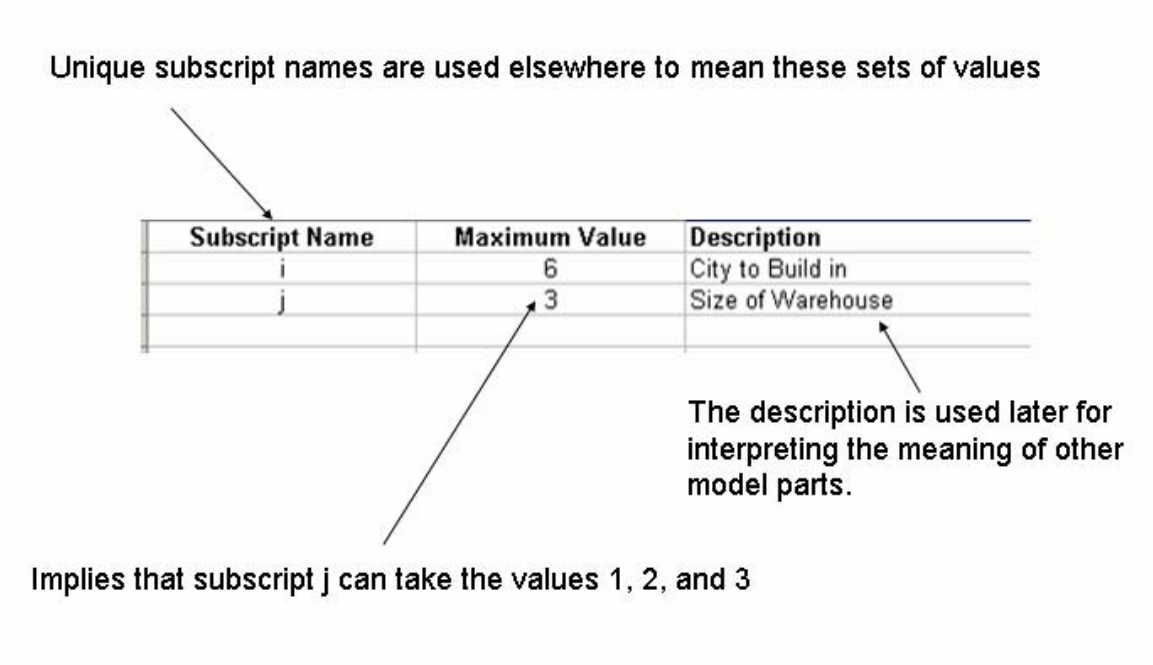
Names, in general, must start with a character (not E) and cannot be more than 10 characters long. In addition, the names "Bound" and "RHSColumn" are reserved for specific components of the MPS formulation and cannot be used to name a subscript, parameter, variable, or constraint.

The following six worksheets are created from templates that are generated with the "Build Model Sheets" option in the Model Builder Menu. The model builder add-in uses the worksheet names to navigate through the model, thus it is important that worksheet names are not changed.

A2.3 Subscript Definition

The subscript definition worksheet contains three columns: subscript name, maximum value, and description. The subscript name is used in subsequent worksheets to refer to this particular subscript. The maximum value specifies the largest integer value that the subscript takes. All subscripts have ranges equal to the set of integers from 1 to the maximum value, inclusive. The description column is used to describe in words what the model means. This description is used elsewhere to aid in interpreting subscripted parameters, variables, and constraints.

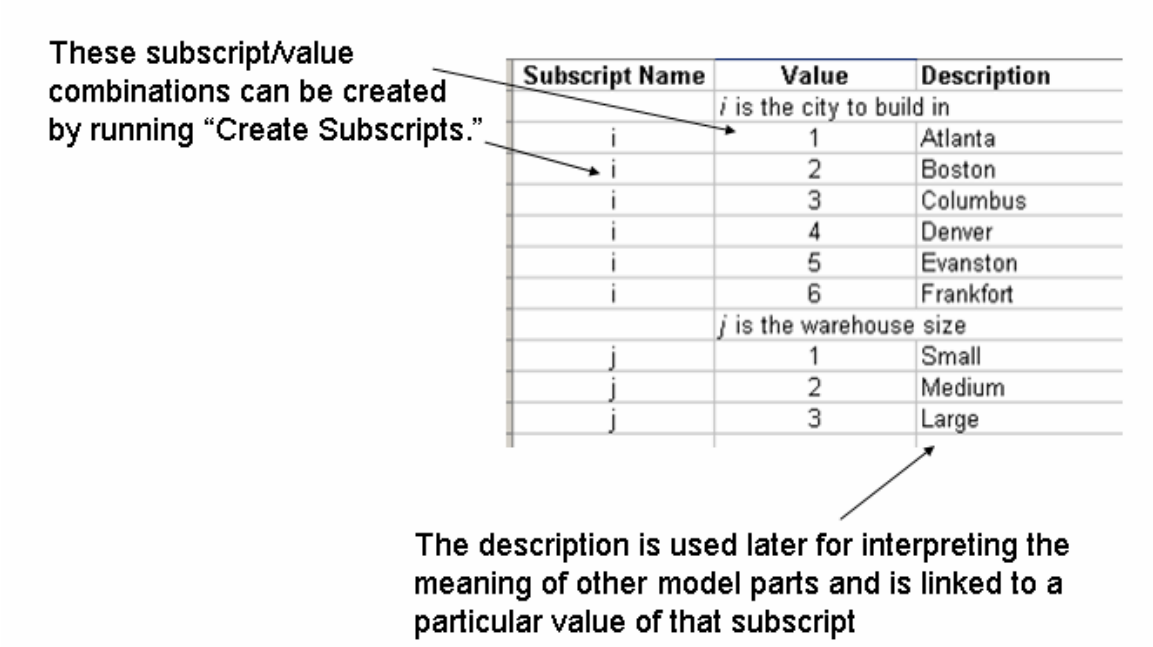
Sample Subscript Definition Worksheet



A2.4 Subscript Values

The Subscript Values worksheet also contains three columns: subscript name, value, and description. Each possible value of each subscript must appear on this worksheet. To reduce the work the user needs to do, a provided menu option, "Create Subscript," is provided which reads the subscript definition worksheet and creates all of the required subscript name and value combinations. The user only needs to add a description for each value of each subscript. This description is used elsewhere to aid in interpreting subscripted parameters, variables, and constraints.

Sample Subscript Values Worksheet

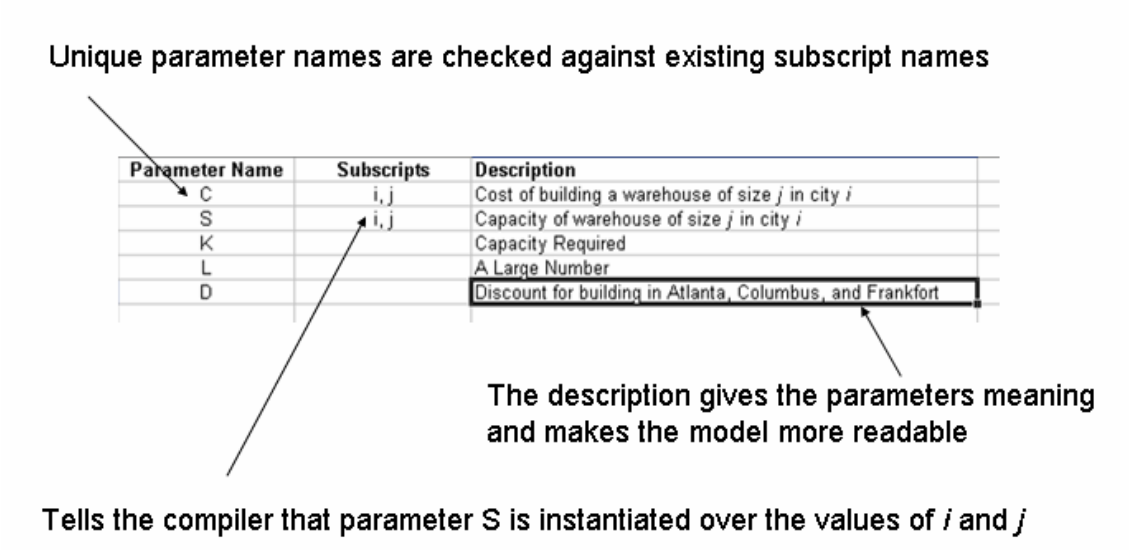


A2.5 Parameter Definition

The parameter definition worksheet is where the user names parameters, the way those parameters are subscripted, and contains space for a description to make the model

more readable. If subscripts are used, the parameter name will be the start of the name used for the final parameter. The end of final names will correspond to the subscript values or, if space does not exist for all the subscript values, a six digit number will be added to the end of the parameter name. In order to index a parameter by a subscript, that subscript is listed in the Subscripts column. In subsequent worksheets, the order of subscripts is assumed to be the order in which subscripts are listed on this worksheet. Multiple subscripts are given as a list with commas between each subscript.

Sample Parameter Definition Worksheet

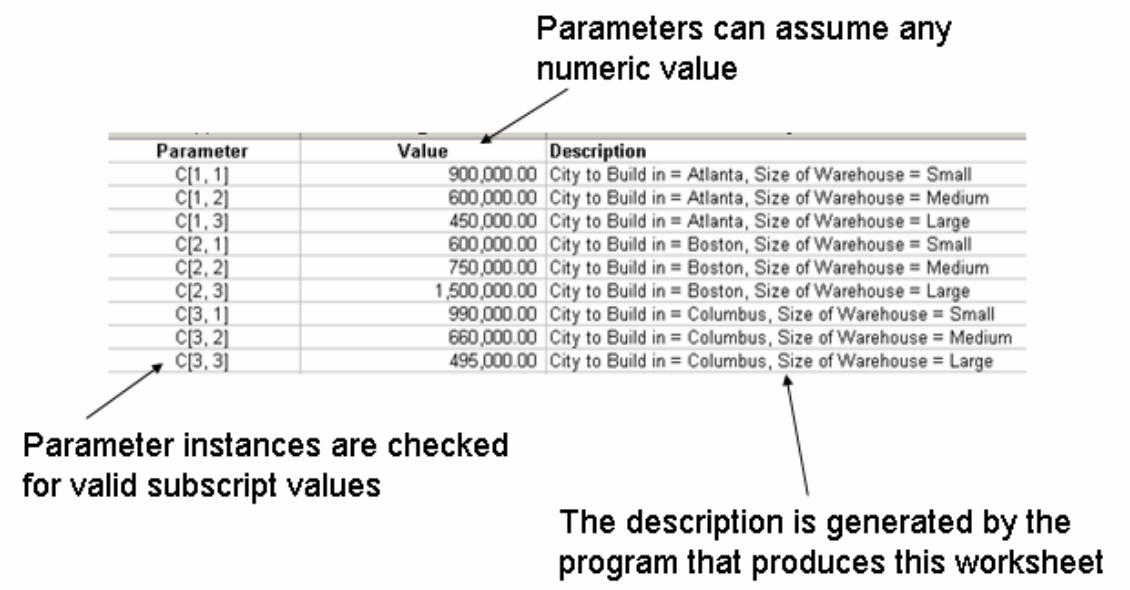


A2.6 Parameter Values

The parameter values worksheet contains each parameter with braces after the name and subscripts replaced with specific subscript values, the value that the particular instance of the parameter will take, and a description. The "Create Parameters" menu option will create all of the final parameter names in the name column and create a description based on the subscript descriptions. The values entered must be numeric and

may be calculated through formulas. If a parameter is omitted from this worksheet, it is assumed to be zero. The program will warn the user when parameters are assumed to be zero. This table is used later to replace parameters with constants.

Sample Parameter Values Worksheet



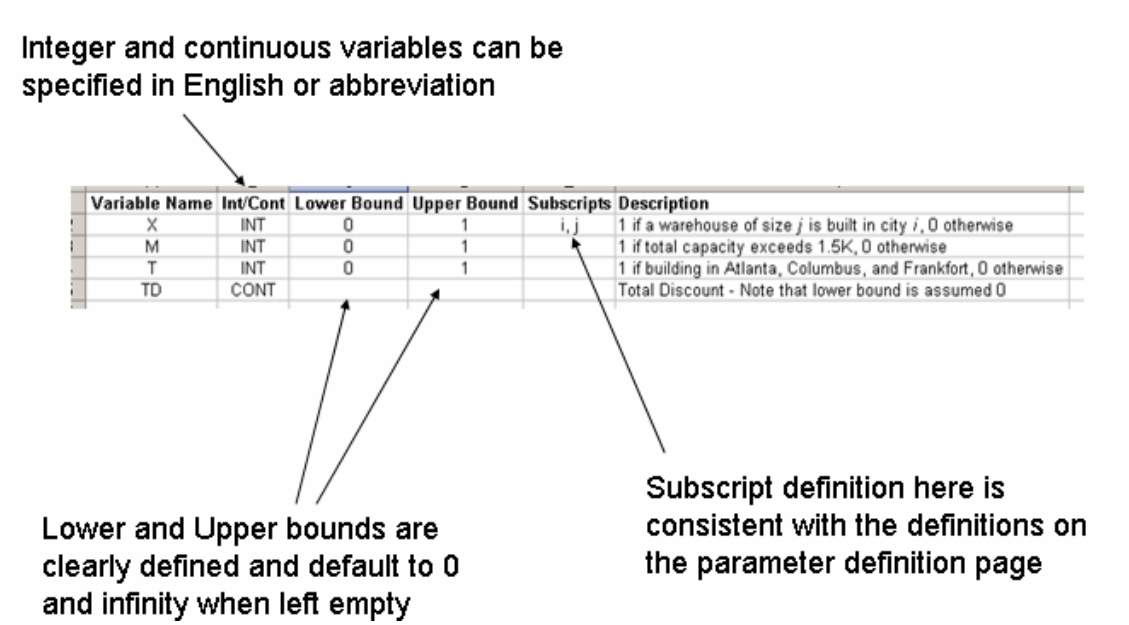
Parameter	Value	Description
C[1, 1]	900,000.00	City to Build in = Atlanta, Size of Warehouse = Small
C[1, 2]	600,000.00	City to Build in = Atlanta, Size of Warehouse = Medium
C[1, 3]	450,000.00	City to Build in = Atlanta, Size of Warehouse = Large
C[2, 1]	600,000.00	City to Build in = Boston, Size of Warehouse = Small
C[2, 2]	750,000.00	City to Build in = Boston, Size of Warehouse = Medium
C[2, 3]	1,500,000.00	City to Build in = Boston, Size of Warehouse = Large
C[3, 1]	990,000.00	City to Build in = Columbus, Size of Warehouse = Small
C[3, 2]	660,000.00	City to Build in = Columbus, Size of Warehouse = Medium
C[3, 3]	495,000.00	City to Build in = Columbus, Size of Warehouse = Large

A2.7 Variable Definition

The variable definition worksheet contains columns for entering the data associated with variables. The variable name column is where the user specifies a unique name for each variable. If subscripts are used, this name will be the start of the names used for the final variables. The end of those names will correspond to the subscript values or, if space does not exist for all the subscript values, a six digit number will be added to the end of the variable name. The Int/Cont column is where the user specifies the type of variable. If the first letter of the type is “I”, the variable will be constrained to integer values. If the first letter of the type is “C”, the variable will be considered a continuous variable. The lower bound is assumed to be zero if not explicitly defined. In

order to ensure valid models, the lower bound must be greater than or equal to zero. The upper bound is assumed to be infinity if not explicitly stated. The upper bound must be greater than the lower bound. The subscripts column is used to list the subscripts that are used to dimension the variable. Leave this column blank to create a scalar variable. A comma should be placed between subscripts. All subscripts are assumed to be in the order defined on this worksheet when the variable is being interpreted elsewhere. The description should be used to increase model readability.

Sample Variable Definition Worksheet



A2.8 Equation Definition

The equation definition worksheet is used to define the objective function and all constraints. The objective function always appears on the first line and has the same structure as the constraint equations with three important differences: constraint repetition cannot be used with the objective function, the Row Sense is Min or Max, and the RHS is

left blank. The row sense for the objective function must start with Min or Max. This allows the user to write out Minimum and Maximum if they so choose. All non-objective equations can include a "for all" or repetition declaration. When this declaration is included, the constraint is repeated so that each of the combinations of variables has its own constraint in the MPS model. Subscripts in the "for all" declaration can have three different types of declarations as shown in the table below.

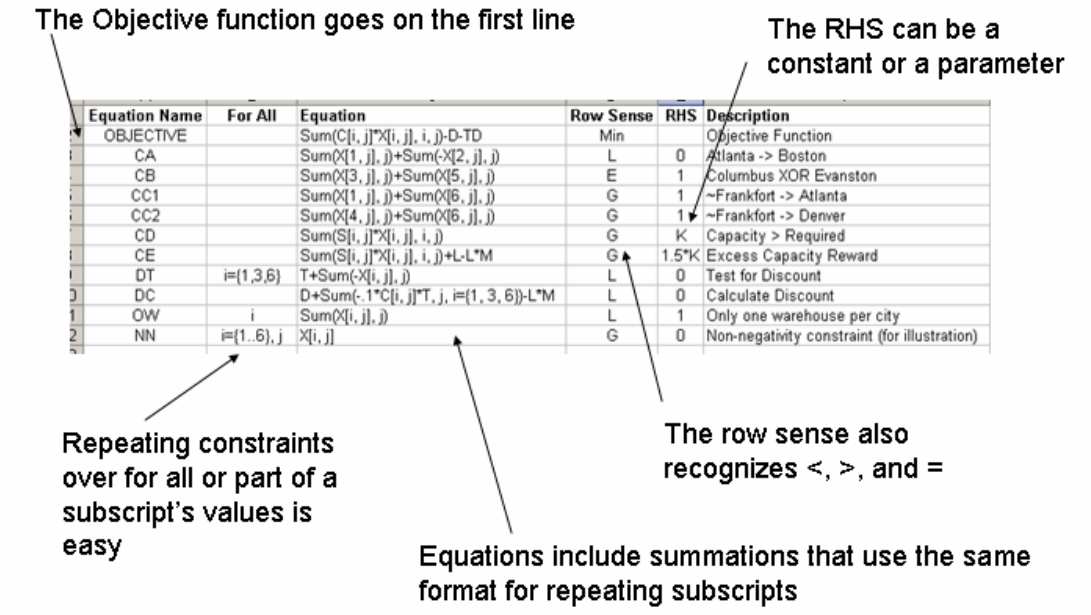
Declaration Example	Interpretation
i	Repeat constraint, substituting in each value of i
i={3}	Replace i with 3
i={1,5,6}	Repeat constraint three times with 1, 5, and 6 substituted for i
i={1..4}	Repeat constraint four times with all integers between 1 and 4 inclusive substituted for i

In constraint equations, the row sense, which tells the program what kind of constraint is in effect, can be input as <, >, =, L, G, E, or any word that starts with L, G, or E (this is not case sensitive). The right hand side (RHS) can be a numeric literal or any combination of parameters. Division in the RHS is not currently supported but multiplication, addition, and subtraction are supported. The description is to aid in the interpretation of the model.

The equation column contains the equation definition for the constraint or objective function. Scalar variables and parameters are referred to by name (i.e. X). Vector variables and parameters are written with a "["directly after the name, a list of the subscripts, and a "]" at the end. The subscripts may be the subscript name or a valid value of that subscript. For example, the 3rd value of X would be written X[3], the ith value of Y would be written as Y[i], and the jth value of the 3rd row of Z would be written as Z[3, j]. Parameters may be multiplied together and multiplied with variables.

Parameters and variables may be added to or subtracted from other parameters and variables. A summation function is also supported. Summations may not be negated and can only be added to other terms. The summation syntax is "sum(" then a linear combination of parameters and variables followed by a comma and a listing of subscripts to sum over. For example, to add together C (a parameter) times X (a variable) for all values of i and j, the declaration would be "Sum(C[i, j]*X[i, j], i, j)." Subscripts can be defined using the same syntax as in the "for all" column. For example, to add together the 1st, 3rd and 5th values of W, the declaration would be "Sum(W[i], i={1,3,5})." Please note that every subscript used in a parameter or variable in an equation must appear in the subscript list of the summation that contains that parameter or variable or the subscript must appear in the "for all" column.

Sample Variable Definition Worksheet



Equation Name	For All	Equation	Row Sense	RHS	Description
OBJECTIVE		Sum(C[i, j]*X[i, j], i, j)-D-TD	Min		Objective Function
CA		Sum(X[1, j], j)+Sum(-X[2, j], j)	L	0	Atlanta -> Boston
CB		Sum(X[3, j], j)+Sum(X[5, j], j)	E	1	Columbus XOR Evanston
CC1		Sum(X[1, j], j)+Sum(X[6, j], j)	G	1	~Frankfort -> Atlanta
CC2		Sum(X[4, j], j)+Sum(X[6, j], j)	G	1	~Frankfort -> Denver
CD		Sum(S[i, j]*X[i, j], i, j)	G	K	Capacity > Required
CE		Sum(S[i, j]*X[i, j], i, j)+L-L*M	G	1.5*K	Excess Capacity Reward
DT	i={1,3,6}	T+Sum(-X[i, j], j)	L	0	Test for Discount
DC		D+Sum(-.1*C[i, j]*T, j, i={1, 3, 6})-L*M	L	0	Calculate Discount
OW	i	Sum(X[i, j], j)	L	1	Only one warehouse per city
NN	i={1..6}, j	X[i, j]	G	0	Non-negativity constraint (for illustration)

A2.9 Creating the MPS file

To create the MPS file, choose the "Create MPS Model" menu option. This creates a copy of the MPS model in a new Excel worksheet called "MPS Formulation." After inspecting and/or changing the MPS model, you may copy and paste the model into a text document or use the "Export to Text File" menu option to export the model to a text file.

When the MPS model is created, variables and constraints that range over subscripts are given new names to indicate that they represent a specific instance of that constraint or variable. This new name will consist of the name of the variable or constraint and the value of each subscript, in the same order as they were entered, with a "_" placed between values. If the name would be over 16 characters, the subscript list is replaced with a six digit number. Additional functionality to accommodate a larger number of constraints and variables and an output that shows how all variables and constraints should be interpreted should be among the first improvements made to the add-in.

Appendix 3: An Example Model

The cheese factory production model from the introduction is used again here, in slightly modified form, to illustrate the new format. This appendix includes screen shots for each of the model worksheets and a screen shot for the final model in the workbook.

Cheese Factory Production Model – Fully Parameterized

Subscripts

$i \in \{1,2,3\}$ = the type of cheese

Parameters

D_i = Demand for cheese type i

P_i = Profit made by making and selling 1 ton of cheese type i

CC_i = Yards of Cheese Cloth used to make 1 ton of cheese type i

M_i = Gallons of milk used to make 1 ton of cheese type i

CCA = Yards of Cheese Cloth Available

MA = Gallons of Milk Available

Variables

C_i = Tons of Cheese Type i Produced

Objective Function

$Max: Z = \sum_i P_i * C_i$ Total profit

Constraints

$C_i \le D_i \quad \forall i$ Limits the cheese produced to demand

$\sum_i CC_i * C_i \le CCA$ Limit based on amount of cheese cloth available

$\sum_i M_i * C_i \le MA$ Limit based on amount of milk available

Cheese Production Model - Subscript Definition

	A	B	C	D
1	**Subscript Name**	**Maximum Value**	**Description**	
2	i	3	Cheese Type	
3				
4				
5				

Cheese Production Model - Subscript Values

	A	B	C	D
1	**Subscript Name**	**Value**	**Description**	
2	i	1	Type 1	
3	i	2	Type 2	
4	i	3	Type 3	
5				
6				

Cheese Production Model - Parameter Definition

	A	B	C	D
1	**Parameter Name**	**Subscripts**	**Description**	
2	D	i	Demand, in tons, for cheese *i*	
3	P	i	Total Profit per ton of cheese *i*	
4	CC	i	Yards of Cheese Cloth used to produce a ton of cheese *i*	
5	M	i	Gallons of Milk used to produce a ton of cheese *i*	
6	CCA		Yards of Cheese Cloth Available for production	
7	MA		Gallons of Milk Available for production	
8				
9				

Cheese Production Model – Parameter Values

	A	B	C	D
1	**Parameter**	**Value**	**Description**	
2	D[1]	35.00	Cheese Type = Type 1	
3	D[2]	30.00	Cheese Type = Type 2	
4	D[3]	40.00	Cheese Type = Type 3	
5	P[1]	1,000.00	Cheese Type = Type 1	
6	P[2]	3,000.00	Cheese Type = Type 2	
7	P[3]	2,500.00	Cheese Type = Type 3	
8	CC[1]	100.00	Cheese Type = Type 1	
9	CC[2]	200.00	Cheese Type = Type 2	
10	CC[3]	-	Cheese Type = Type 3	
11	M[1]	600.00	Cheese Type = Type 1	
12	M[2]	900.00	Cheese Type = Type 2	
13	M[3]	800.00	Cheese Type = Type 3	
14	CCA	4,500.00	Yards of Cheese Cloth Available for production	
15	MA	35,000.00	Gallons of Milk Available for production	
16				

Cheese Production Model - Variable Definition

	A	B	C	D	E	F
1	**Variable Name**	**Integer/Continuous**	**Lower Bound**	**Upper Bound**	**Subscripts**	**Description**
2	C	Continuous			i	Tons of Cheese *i* produced
3						
4						

Cheese Production Model - Equation Definition

	A	B	C	D	E	F
1	**Equation Name**	**For All**	**Equation**	**Row Sense**	**RHS**	**Description**
2	OBJECTIVE		sum(P[i]*C[i], i)	Max		Objective Function - Do Not Move From Row 2
3	Demand	i	C[i]	<	D[i]	Don't produce more cheese than demand
4	ChCloth		sum(CC[i]*C[i], i)	<	CCA	Limit placed on production by cheese cloth availablity
5	Milk		sum(M[i]*C[i], i)	<	MA	Limit placed on production by availablity of milk
6						
7						

Cheese Production Model – MPS Formulation (Created by Excel Add-In)

	A	B	C	D	E	F
2	OBJSENSE					
3		MAXIMIZE				
4	ROWS					
5		N	OBJECTIVE			
6		L	Demand_1			
7		L	Demand_2			
8		L	Demand_3			
9		L	ChCloth			
10		L	Milk			
11	COLUMNS					
12		C[1]	OBJECTIVE	1000		
13		C[1]	Demand_1	1		
14		C[1]	ChCloth	100		
15		C[1]	Milk	600		
16		C[2]	OBJECTIVE	3000		
17		C[2]	Demand_2	1		
18		C[2]	ChCloth	200		
19		C[2]	Milk	900		
20		C[3]	OBJECTIVE	2500		
21		C[3]	Demand_3	1		
22		C[3]	ChCloth	0		
23		C[3]	Milk	800		
24	RHS					
25		RHSColumn	Demand_1	35		
26		RHSColumn	Demand_2	30		
27		RHSColumn	Demand_3	40		
28		RHSColumn	ChCloth	4500		
29		RHSColumn	Milk	35000		
30	BOUNDS					
31		LO	Bound	C[1]	0	
32		LO	Bound	C[2]	0	
33		LO	Bound	C[3]	0	
34	ENDATA					